A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding

David DeHaan, David Toman, Mariano P. Consens, M. Tamer Özsu
University of Waterloo
School of Computer Science
Waterloo, Canada
{dedehaan, david, mconsens, tozsu}@uwaterloo.ca

ABSTRACT

The W3C XQuery language recommendation, based on a hierarchical and ordered document model, supports a wide variety of constructs and use cases. There is a diversity of approaches and strategies for evaluating XQuery expressions, in many cases only dealing with limited subsets of the language. In this paper we describe an implementation approach that handles XQuery with arbitrarily-nested FLWR expressions, element constructors and built-in functions (including structural comparisons). Our proposal maps an XQuery expression to a single equivalent SQL query using a novel *dynamic interval* encoding of a collection of XML documents as relations, augmented with information tied to the query evaluation environment. The dynamic interval technique enables (suitably enhanced) relational engines to produce predictably good query plans that do not preclude the use of sort-merge join query operators. The benefits are realized despite the challenges presented by intermediate results that create arbitrary documents and the need to preserve document order as prescribed by semantics of XQuery. Finally, our experimental results demonstrate that (native or relational) XML systems can benefit from the above technique to avoid a quadratic scale up penalty that effectively prevents the evaluation of nested FLWR expressions for large documents.

1. INTRODUCTION

With the widespread adoption of XML both as a document format and as a data exchange format, the interest in querying growing XML repositories has increased. XQuery [10] is emerging as the standard XML query language and there is growing recognition that it can rapidly gain a high level of adoption. The language addresses a wide range of requirements, thus incorporating a rich set of features [17].

A diverse set of strategies for evaluating queries over XML data has been proposed. One approach favors the implementation of XML specific query processors [7, 27]. Other approaches attempt to leverage relational implementations by resorting to encoding XML data as relations and translating XML queries into relational queries [9, 29, 32]. Such a translation is particularly relevant when the XML data are already stored in a relational system. This may happen because relational data are used to produce XML documents, or because XML documents are stored in a relational database, or a combination of both. Even when the source XML is external to the relational system, there are still very good reasons to use a SQL engine as the target query processor for XQuery expressions: maturity of the implementations, extensive tuning, proven scalability, sophisticated optimizers, etc.

Example 1.1 We use the query "*list names of persons and the number of items they bought*" (Q8) from the XMark benchmark [6] as a running example throughout the paper:

```
for $p in document("auction.xml")/site/people/person
let $a := for $t in document("auction.xml")/site/
                        closed_auctions/closed_auction
          where $t/buyer/@person = $p/@id
          return $t
return <item person="{$p/name/text()}">count($a)</item>
```

This query shows nested FLWR (for-let-where-return) expression and is used, in this paper, to illustrate many of the features of the proposed translation.

Sample data used by the running example is taken from the relevant portion of an XMark document and is shown in Figure 1.◇

There are significant challenges in evaluating XQuery expressions using a SQL query processor. The most commonly cited difficulties connected with the use of relational technology are often linked to the following features of XQuery expressions:

1. The iterative nature of many XML operations, e.g., the *ancestor/descendent/sibling* like operations, combined with the nested nature of XML documents;
2. The definition and use of *structural equality* in FLWR `where` clauses;
3. The full compositionality of XML query expressions, in particular the possibility of nesting FLWR expressions within functions operating on XML documents (cf. Figures 2 and 3);

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

```
<site>
 <people>
  <person id="person0">
   <name>Jaak Tempesti</name>
   <emailaddress>mailto:Tempesti@labs.com</emailaddress>
   <phone>+0 (873) 14873867</phone>
   <homepage>http://www.labs.com/~Tempesti</homepage>
  </person>
  <person id="person1">
   <name>Cong Rosca</name>
   <emailaddress>mailto:Rosca@washington.edu</emailaddress>
   <phone>+0 (64) 27711230</phone>
   <homepage>http://www.washington.edu/~Rosca</homepage>
  </person>
  ...
 </people>
 <closed_auctions>
  <closed_auction>
   <seller person="person0" />
   <buyer person="person1" />
   <itemref item="item1" />
   <price>42.12</price>
   <date>08/22/1999</date>
   <quantity>1</quantity>
   <type>Regular</type>
  </closed_auction>
  ...
 </closed_auctions>
 ...
</site>
```

Figure 1: Portion of XMark Database

4. The need for *constructed XML documents* representing intermediate results during query evaluation;
5. The translation of *sorting within XML queries*; and
6. The requirement to preserve the *document order* [16].

The above features are often at odds with the *first-order* and *first normal form* nature of relational queries. Indeed, queries utilizing various combinations of the above features are often labeled as *untranslatable* to SQL and an escape to a general-purpose programming language is commonly used to fully support these features [26]. In addition, the requirement to evaluate queries while preserving the document order has often been used as a justification for restricting implementation to *nested-loop* style strategies. Indeed, our experimental results show that most current implementations [1, 2, 3, 4, 5] exhibit undesirable (at least) quadratic behavior consistent with the use of such strategies.

In this paper we provide a novel solution to all of these challenges based on an encoding of XML documents using *dynamic intervals*, an extension of the standard encoding of ordered forests based upon intervals (or regions) [15, 20, 21]. The extended encoding allows us to represent an *iterated* application of XQuery expressions on a sequence of XML documents by a single relational query. The contributions of our approach and of this paper are:

1. We define a novel encoding of sequences of XML documents using *dynamic intervals*. Although interval encoding of trees has been discussed before [15], the use of *dynamic* intervals to handle iteration constructs in XQuery is novel to this paper. Approaches that maintain a static interval for each element throughout the entire query evaluation cannot handle nested queries (such as Q9 in Section 6.3) due to document order constraints on the results of nested FLWR expressions. Furthermore, dynamic intervals are essential for the compositionality of translated XQuery expressions.
2. We develop a *comprehensive* translation of XQuery (including XPath) expressions to relational queries operating on this encoding. The proposed translation is *compositional* and fully supports arbitrary (syntactically valid) combinations and nesting of basic functions and FLWR expressions (cf. Figures 2 and 3) without the need for an escape to a general-purpose programming language. This automatically guarantees termination of queries and low data complexity of query evaluation. The space of XQuery expressions translatable to a single SQL statement is significantly richer with this technique than that handled by previous proposals (e.g., [26]).
3. The proposed approach provides an implementation strategy for the generated queries on top of standard relational technology. In particular, the translation does not impose any restrictions on the use of computationally preferable query operators, e.g., the use of *merge-sort joins*, in the generated queries while still adhering to the *semantics* of XQuery, in particular to the requirement to *preserve document ordering*.
4. Performance advantages of our approach are supported by experimental results. There is no comparable analysis to the one presented in this paper that shows that predictably good query plans could be generated for the SQL produced by a generic XQuery translation working with arbitrary XML documents.

While the strategies are described in the context of a relational implementation, it is clear that a native XML system can make use of them by manipulating dynamic intervals as an internal representation. The proposed XQuery processor assumes interval encoded XML forests on input. At first glance, this may suggest potential update problems; however, the issue of updates is orthogonal to the strategies developed in this paper and can be handled by techniques described in [18, 19, 31].

The rest of the paper is organized as follows: Section 2 provides a concise but sufficiently precise definition of an XQuery fragment used throughout the paper to illustrate the capabilities and features of our approach. Note that all features of XQuery's FLWR expressions including XPath can be handled in this language[1]. Section 3 introduces *dynamic intervals*, an interval-based relational encoding for XML documents augmented with information tied to the query evaluation environment. The translation of an XQuery expression into a *single SQL expression* is given in Section 4, fully accounting for arbitrarily-nested FLWR expressions, element constructors, and a wide variety of built-in functions for dealing with document structure and order. While commercial SQL engines can be used to evaluate the resulting SQL queries produced by the translation, there are areas where performance can be further improved (mainly because commercial relational engines are tuned to a different query workload). Consequently, Section 5 describes additional relational query plan operators that fully exploit properties of ordered data sets during query execution. The techniques presented in the paper enable query execution plans based on sorting and linear merge-join operations even in

[1]Similar to most database approaches, we do not handle general recursive functions allowed by the XQuery proposal.

Constructors:	$[\,] : \mathsf{XF}$	the empty forest constructor
	$\mathsf{XNode} : \mathsf{String} \times \mathsf{XF} \to \mathsf{XF}$	the element constructor (adds a labeled root to a forest)
	$@ : \mathsf{XF} \times \mathsf{XF} \to \mathsf{XF}$	the concatenation operator
Horizontal Operations:	$\mathsf{head} : \mathsf{XF} \to \mathsf{XF}$	first element of a forest
	$\mathsf{tail} : \mathsf{XF} \to \mathsf{XF}$	all but the first element of the forest
	$\mathsf{reverse} : \mathsf{XF} \to \mathsf{XF}$	the forest in reverse order (top-level only)
	$\mathsf{select} : \mathsf{String} \times \mathsf{XF} \to \mathsf{XF}$	subforest of trees with 1st arg value in their root
	$\mathsf{distinct} : \mathsf{XF} \to \mathsf{XF}$	subforest of distinct trees (1st preserved)
	$\mathsf{sort} : \mathsf{XF} \to \mathsf{XF}$	a forest ordered by *tree order*
Vertical Operations:	$\mathsf{roots} : \mathsf{XF} \to \mathsf{XF}$	a forest of root nodes
	$\mathsf{children} : \mathsf{XF} \to \mathsf{XF}$	a forest of all children in original order
	$\mathsf{subtreesdfs} : \mathsf{XF} \to \mathsf{XF}$	a forest of all subtrees in DFS order
Boolean Conditions:	$\mathsf{equal} : \mathsf{XF} \times \mathsf{XF} \to \mathsf{Bool}$	test for structural (tree) equality
	$\mathsf{less} : \mathsf{XF} \times \mathsf{XF} \to \mathsf{Bool}$	structural (tree) ordering
	$\mathsf{empty} : \mathsf{XF} \to \mathsf{Bool}$	test for emptiness

Figure 2: Sample Basic Operations on XML Forests.

$$
\begin{array}{rcl}
[\![\, x \,]\!]\, E & = & E(x) \\
[\![\, \mathsf{XFn}(e_1, \ldots, e_k) \,]\!]\, E & = & [\![\, \mathsf{XFn} \,]\!]([\![\, e_1 \,]\!]\, E, \ldots, [\![\, e_k \,]\!]\, E) \\
[\![\, \mathsf{let}\, x = e\, \mathsf{in}\, e' \,]\!]\, E & = & [\![\, e' \,]\!](E[x := ([\![\, e \,]\!]\, E)]) \\
[\![\, \mathsf{where}\, \varphi\, \mathsf{return}\, e \,]\!]\, E & = & \text{if } [\![\, \varphi \,]\!]\, E \text{ then } [\![\, e \,]\!]\, E \text{ else } [\,] \\
[\![\, \mathsf{for}\, x \in e\, \mathsf{do}\, e' \,]\!]\, E & = & [\![\, e' \,]\!](E[x := v_1]) @ \ldots @\, [\![\, e' \,]\!](E[x := v_k]) \text{ where } [v_1, \ldots, v_k] = [\![\, e \,]\!]\, E
\end{array}
$$

Figure 3: Semantics of FLWR-like Expressions.

the presence of nested FLWR constructs. Section 6 provides experimental data that support our claims. These experiments, conducted using the XMark benchmark, demonstrate that iteration operations of XQuery can be efficiently implemented using the approach proposed in this paper, which scales similar to relational sort-merge join algorithms. This is in contrast to all the other XQuery processors that we have tested where the scale-up is quadratic. Related work is discussed in Section 7. We conclude in Section 8.

2. XQUERY IN A NUTSHELL

The goal of this section is to provide succinct definitions for basic XML documents and a query language that captures the complexities of dealing with the full XQuery language but is more amenable to analysis.

We first introduce a simple model for an XML document as an ordered forest of rooted, node-labeled, ordered trees.

Definition 2.1 (XML Forests) We define the set XF of XML forests inductively by

$$\mathsf{XF} \quad = \quad [\,] \;\mid\; [\,\texttt{<s>}\,\mathsf{XF}\,\texttt{</s>}\,] \;\mid\; \mathsf{XF} \,@\, \mathsf{XF}$$

where s is a String, $[\,]$ denotes an empty forest, $[\,\texttt{<s>}\,\mathsf{XF}\,\texttt{</s>}\,]$ denotes a forest containing a single tree with a root labeled s and an ordered forest XF as children, and $\mathsf{XF} \,@\, \mathsf{XF}$ denotes the concatenation of two ordered forests.

The preceding definition has no explicit accounting of node identity, node typing, "don't care" child node order, and so on. Such features, however, can be easily added by additional encoding conventions that relate either to node labeling or to subtree patterns. A text leaf node with CDATA `text` can be encoded using the label "`text`", while an element `tag` is encoded using the label "`<tag>`". A similar approach can be taken to represent attributes, and so on.

Definition 2.2 (Minimal XQuery) The syntax for a XQuery-like language is given by the following BNF rule:

$$
\begin{array}{rcl}
e & ::= & x \\
& \mid & \mathsf{XFn}(e_1, \ldots, e_k) \\
& \mid & \mathsf{let}\, x = e\, \mathsf{in}\, e' \\
& \mid & \mathsf{where}\, \varphi\, \mathsf{return}\, e \\
& \mid & \mathsf{for}\, x \in e\, \mathsf{do}\, e'
\end{array}
$$

where e, e', and $e_1, \ldots, e_k$ are expressions, x is a variable, XFn a function on XML forests, and φ is a boolean condition introduced in Figure 2.

In Figure 3, the semantics of these expressions is defined inductively with respect to an *environment* E using a denotational semantics definition of the $[\![\,.\,]\!]$ map. The environment E is used to supply values (XML forests) for free variables in an expression; the function E maps names of (defined) variables to XML forests ("stored" in these variables by the $\mathsf{let}\, x = e\, \mathsf{in}\, e'$ and $\mathsf{for}\, x \in e\, \mathsf{do}\, e'$ constructs or as names for input documents).

The language defined above can be seen as a reduced version of the XQuery core language employed to formalize the semantics of XQuery [22]. Despite its simplicity, the language still captures all the intricate nuances present in the XQuery's FLWR and XPath expressions. The syntax allows arbitrary composition of basic function invocations, local variable definitions, filtering by Boolean conditions and iteration over the trees in XML forests[2].

3. DYNAMIC INTERVALS

To manipulate XML documents in a relational system, we need to select an *encoding*—that is, a relational representation that captures enough information about XML to map

[2]Some of the constructs, e.g. the `where` clause, are not allowed to stand alone in XQuery. However, this is irrelevant to the development in this paper, and our approach indeed handles proper XQuery syntax used in the examples.

s	l	r
<site>	0	85
<people>	1	46
<person>	2	23
@id	3	6
person0	4	5
<name>	7	10
Jaak Tempesti	8	9
⋮	⋮	⋮

Figure 4: ENCODED XMARK DOCUMENT

documents to and from relations. We do this in two stages. First we encode an XML forest as triples: the node label plus the right and left endpoints of the intervals associated with the node.

Definition 3.1 (Interval Encoding) Let $f \in \mathsf{XF}$ be an XML forest. We define an instance of a ternary relation $X \subseteq \mathsf{String} \times \mathsf{Nat} \times \mathsf{Nat}$ to contain a tuple (s, l, r) for every node labeled s (node s, in short) in the forest f. The values l and r are constrained as follows:

- $l < r$ for all $(s, l, r) \in X$,
- if node s_1 is an ancestor of node s_2 in f then the corresponding tuples $(s_1, l_1, r_1) \in X$, and $(s_2, l_2, r_2) \in X$ satisfy $l_1 < l_2$ and $r_2 < r_1$, and
- if the node s_1 is a left sibling of the node s_2 in f then the tuples $(s_1, l_1, r_1) \in X$ and $(s_2, l_2, r_2) \in X$ satisfy $r_1 < l_2$.

We say that the (instance of the) relation X *represents* (is an encoding for) the XML forest f. We associate a value $w \in \mathsf{Nat}$ that satisfies the condition $w > \max\{r : (s, l, r) \in X\}$ with a fixed representation X of an XML forest f, and we call w the *width* of (the representation of) f.

The definition of the interval encoding is a very general one that captures several more specific cases, in particular the encoding where the endpoints denote the offsets in a string representation of the XML document. Note that we do not require the intervals representing the elements in a document nor the width w of a document to be *tight*; indeed any values that satisfy the above inequalities are acceptable.

Example 3.2 One way to generate a valid interval encoding is to perform a depth-first traversal of the document tree, using an incrementing counter that assigns the l value to a node when it is first encountered, and the r value when the node is seen for the last time. Figure 4 shows the relation resulting from applying this algorithm to the data in Figure 1. This encoding has a width of 86. ⋄

The second stage in the definition of our encoding is to extend the representation of individual XML forests to capture a sequence of environments (introduced in Figure 3). The goal is to simulate the evaluation of arbitrary FLWR expressions using a fixed relational query.

In particular, for the case of the "$\mathsf{for}\, x \in e\, \mathsf{do}\, e'$" expression, the semantic equation defines an iterator over a sequence of such environments; the result is then *the concatenation* of the results from the individual iterations. To simulate these semantics using a single relational query we need a relational representation of *all environments* of the form $E[x := v_i]$ that participate in the evaluation of the iterated expression (cf. Figure 3). The general definition is as follows:

Definition 3.3 (Dynamic Intervals) Let $[E_1, \ldots, E_n]$ be a sequence of environments for variables $x_1, \ldots, x_m$ of the form $E_i = \langle x_1 = f_1^i, \ldots, x_m = f_m^i \rangle$, where f_j^i are XML forests. We say that an instance of a relational schema $I, T_{x_1}, \ldots, T_{x_m}$, where $I \subseteq \mathsf{Nat}$ and $T_{x_i} \subseteq \mathsf{String} \times \mathsf{Nat} \times \mathsf{Nat}$, *represents* $[E_1, \ldots, E_n]$ if

- $I = \{i_1, \ldots, i_n\}$ such that $i_k < i_l$ whenever $k < l$, and
- $T_{x_j} = \bigcup_{k=1}^{n} \{(s, l + i_k w_{x_j}, r + i_k w_{x_j}) : (s, l, r) \in X_j^k\}$ where X_j^k represents f_j^k

for $w_{x_j} = max\{w : w \text{ width of } X_j^k, 1 \leq k \leq n\}$. We call w_{x_j} the *width* of (the representation of) x_j.

In essence, we have created an encoding for a sequence of tuples of XML forests representing the values for variables (i.e., environments), together with an *index set* that tells us what range the values bound to variables occupy in the overall interval encoding. The dynamic interval representation serves the dual purpose of maintaining the separation of an arbitrary sequence of environments while, at the same time, being able to be interpreted as the representation for the single forest that is the result of concatenating all the environments in the sequence (the second use allows us to *exit* a $\mathsf{for}\, x \in e\, \mathsf{do}\, e'$ loop without any additional computation needed to concatenate the results). At query translation time the intervals and environment sizes are dynamically allocated with varying widths to maintain the separation of XML forests belonging to each of the environments.

Example 3.4 Consider the first path expression in query Q8 from Example 1.1:

```
document("auction.xml")/site/people/person
```

Figure 5 shows the index relation, I, for the initial query environment, together with the results of the path expression, $T_{\texttt{person}}$, within this initial environment. The path expression has a (worst case) width $w_{\texttt{person}} = w_{\texttt{document}} = 86$ (see Section 4.1).⋄

4. SQL TRANSLATION FOR XQUERY

The translation of XQuery expressions to relational queries is presented in several steps. The first step consists of translations for the basic operations on XML forests outlined in Figure 2. The remaining steps describe a fully compositional translation for FLWR expressions that mirror the semantic rules in Figure 3.

The translation is presented as a sequence of SQL *templates* for fragments of queries that are composed to produce the final query. Each template can be understood as a relational view with table (query) parameters that are to be substituted by other templates/views. We use the notation

```
CREATE VIEW V(T1,...,Tk) AS Q(T1,...,Tk)
```

to stand for a template with parameters `T1`,...,`Tk`.

I	T_{person}		
i	s	l	r
0	<person>	2	23
	@id	3	6
	person0	4	5
	<name>	7	10
	Jaak Tempesti	8	9
	<emailaddress>	11	14
	mailto:Tempesti@labs.com	12	13
	<phone>	15	18
	+0 (873) 14873867	16	17
	<homepage>	19	22
	http://www.labs.com/~Tempesti	20	21
	<person>	24	45
	@id	25	28
	person1	26	27
	<name>	29	32
	Cong Rosca	30	31
	<emailaddress>	33	36
	mailto:Rosca@washington.edu	34	35
	<phone>	37	40
	+0 (64) 27711230	38	39
	<homepage>	41	44
	http://www.washington.edu/~Rosca	42	43
	$w_p = 86$		

Figure 5: Persons in an Initial Environment

4.1 Operators

Each operator on XML forests introduced in Figure 2 has an associated template in the translation. For example, the template for the roots operator is defined as

```
CREATE VIEW ROOTS(T) AS
  SELECT u.s AS s, u.l AS l, u.r AS r
  FROM   T u
  WHERE NOT EXISTS (
    SELECT *
    FROM   T v
    WHERE  v.l < u.l AND u.r < v.r )
```

and the template for the children operator as

```
CREATE VIEW CHILDREN(T) AS
  SELECT u.s AS s, u.l AS l, u.r AS r
  FROM   T u
  WHERE EXISTS (
    SELECT *
    FROM   T v
    WHERE  v.l < u.l AND u.r < v.r )
```

The remaining operators are defined using similar definitions and are omitted for space reasons. For each of the XFn operations we can provide a function w_{XFn} that provides the (upper bound on the) width of the relational representation generated by the template as a function of the widths of the template's inputs. Hence, we have that $w_{[]} = 0$, $w_{\mathsf{XNode}} = w_x+2$, $w_{@} = w_{x_1}+w_{x_2}$, $w_{\mathsf{head}} = w_x$, $w_{\mathsf{tail}} = w_x$, $w_{\mathsf{reverse}} = w_x$, $w_{\mathsf{distinct}} = w_x$, $w_{\mathsf{roots}} = w_x$, $w_{\mathsf{children}} = w_x$, $w_{\mathsf{subtreesdfs}} = w_x^2$, and so on. The *dynamic interval* encoding allows the use of fixed compile-time definitions for the (upper bounds on the) widths of the resulting documents and thus reduce the need to recompute the actual widths during query execution.

Yet another operator template, the element constructor used in a fragment of our running example, is shown below.

Example 4.1 Consider the final `return` statement in Example 1.1. Suppose the view T contains partially constructed results made up of the person attribute label (`@person`), person attribute value, and the value for `COUNT($a)`. The template defining the relational view for the constructor of the `<item>` tag around the contents of T is defined as

```
CREATE VIEW XNODE_item AS
  ( SELECT '<item>' AS s, 0 AS l, 91 AS r
    FROM   UNIT )
  UNION ALL
  ( SELECT s, l+1 as l, r+1 as r
    FROM   T )
```

where `UNIT` is a table containing a single tuple (which is used to construct constant tuples in SQL). The resulting width of each constructed `item` element is 92 computed as follows: `$p/name/text()` has worst case width of 86 plus 2 for each of `@person`, `<item>` and `count($a)`.◇

4.2 FLWR Expressions

The templates for the basic operators in the previous section are defined for single XML forests. This section builds upon the templates for the XFn operators to define templates for the FLWR constructs which evaluate the XFn operator separately for each XF corresponding to a separate environment. In what follows, we assume that $I, T_{x_1}, \ldots, T_{x_m}$ represents the sequence of environments $[E_1, \ldots, E_n]$ as in Definition 3.3. The construction is extended to the FLWR expression e; the pair I, T_e then represents the sequence of forests obtained by evaluating e in $[E_1, \ldots, E_n]$. The remaining subsections provide the definitions for the views T_e (inductively on the structure of e).

4.2.1 Functions

Given XFn, an m-ary operator on XF represented by a relational template $Q_{\mathsf{XFn}}(T_{e_1}, \ldots, T_{e_m})$ (as defined in Section 4.1), we define a template

```
CREATE VIEW T_XFn(e1,...,em)(Te1, ..., Tem)  AS
 SELECT s, l + i * w_XFn AS l, r + i * w_XFn AS r
 FROM   I, Q_XFn (
  ( SELECT s, l - i * we1, r - i * we1
    FROM   Te1
    WHERE  i * we1 ≤ l  AND r < (i + 1) * we1 ),
    ...,
  ( SELECT s, l - i * wem, r - i * wem
    FROM   Tem
    WHERE  i * wem ≤ l  AND r < (i + 1) * wem )
 )
```

where w_{e_i} are the widths of the template's inputs and w_{XFn} is the width of the output. The pair of tables $I, T_{\mathsf{XFn}(e_1,\ldots,e_m)}$ then represents $[\mathsf{XFn}(t_1^1, \ldots, t_m^1), \ldots, \mathsf{XFn}(t_1^n, \ldots, t_m^n)]$.

Example 4.2 Continuing with fragments of our running example, the `<item>`e`</item>` constructor compiles to

```
CREATE VIEW T_XNODE_item AS
  SELECT s, l+i*92 AS l, r+i*92 AS r
  FROM   I,
    ( SELECT '<item>' AS s, 0 AS l, 91 AS r
      FROM   UNIT
    UNION ALL
      SELECT s, l+1 AS l, r+1 AS r
      FROM
      ( SELECT s, l-i*90 AS l, r-i*90 AS r
        FROM   T_e
        WHERE  i*90<=l AND r<(i+1)*90
      )
    )
```

for $w_{\mathsf{e}} = 90$ and $w_{\mathsf{XNode}} = w_e + 2 = 92$, where e is the expression defining the "partial constructed results" discussed in Example 4.1, and `T_e` is the view containing the results of this expression. Lines 4 to 13 in the SQL code above correspond to the expanded template for the constructor translation given that example, in which lines 9 to 12 correspond to expansion of table `T`.◇

4.2.2 Assignment

Manipulating the environment is at the core of the translation of the $\mathsf{let}\, x = e \,\mathsf{in}\, e'$ assignment expression: this expression increases the size of the environment by adding the binding for $x = e$ before executing e'. Therefore, we first *extend* the relational representation of the sequence of environments. The new environment encoding is defined as follows:

```
CREATE VIEW I' AS  SELECT * FROM I
CREATE VIEW T'_{e_i}  AS  SELECT * FROM T_{e_i}
CREATE VIEW T'_x  AS  SELECT * FROM T_e
```

We then apply the translation of e' with respect to this set of (primed) views I',T'_{e_i}, and T'_x obtaining $T'_{e'}$. Finally, we define the result as

```
CREATE VIEW T_{let x=e in e'}  AS  SELECT * FROM T'_{e'}
```

and $w_{\mathsf{let}\, x=e \,\mathsf{in}\, e'} = w_{e'}$.

4.2.3 Conditional

Similar to the translation of the assignment, we first define a representation $I', T'_{e_1}, \ldots, T'_{e_m}$ of a subsequence of $[E_1, \ldots, E_n]$ in which each environment satisfies condition φ. The new set of environment indices is defined by

```
CREATE VIEW I'(T_{e_1}, ..., T_{e_m}) AS
  SELECT i
  FROM   I
  WHERE   EXISTS Q_φ (
    ( SELECT s, l − i ∗ w_{e_1}, r − i ∗ w_{e_1}
      FROM   T_{e_1}
      WHERE  i ∗ w_{e_1} ≤ l  AND r < (i + 1) ∗ w_{e_1} ),
      ···,
    ( SELECT s, l − i ∗ w_{e_m}, r − i ∗ w_{e_k}
      FROM   T_{e_m}
      WHERE  i ∗ w_{e_m} ≤ l  AND r < (i + 1) ∗ w_{e_m} )
  )
```

where Q_φ is a translation of φ and the relations representing the expressions e_i by

```
CREATE VIEW T'_{e_i}  AS
  SELECT s, l, r
  FROM T_{e_i}, I'
  WHERE i ∗ w_{e_i} ≤ l  AND r < (i + 1) ∗ w_{e_i}
```

We use this this representation to define T'_e as

```
CREATE VIEW T_{where φ return e}  AS  SELECT * FROM T'_e
```

where $w_{\mathsf{where}\, \varphi \,\mathsf{return}\, e} = w_e$.

4.2.4 Iteration

We define $I', T'_{e_1}, \ldots, T'_{e_m}, T'_x$ to represent the sequence of environments

$$\left[E_1[x = t_1^{l_1^1}], \ldots, E_1[x = t_1^{l_1^{k_1}}], \ldots, E_n[x = t_n^{l_n^1}], \ldots, E_n[x = t_n^{l_n^{k_n}}]\right]$$

for $[t_i^{l_i^1}, \ldots, t_i^{l_i^{k_i}}] = [\![e]\!] E_i$, $0 < i \leq n$. We then apply the translation of the expression e' on this sequence as required by the semantics of the $\mathsf{for}\, x \in e \,\mathsf{do}\, e'$ iterator expression. The representation is defined as follows:

```
CREATE VIEW I'(T_e) AS
  SELECT i ∗ w_e + r.l AS i
  FROM   I, ROOTS(T_e) r
  WHERE  i ∗ w_e ≤ r.l AND r.r < (i + 1)w_e

CREATE VIEW T'_x(T_e) AS a
  SELECT s, x.l − i ∗ w_e + (i ∗ w_e + r.l) ∗ w_e AS l,
            x.r − i ∗ w_e + (i ∗ w_e + r.l) ∗ w_e AS r
  FROM   I, T_e x, ROOTS(T_e) r
  WHERE  i ∗ w_e ≤ r.l AND r.r < (i + 1)w_e
    AND  r.l ≤ x.l AND x.r ≤ r.r

CREATE VIEW T'_{e_i}(T_{e_i}, T_e) AS a
  SELECT s, x.l − i ∗ w_{e_i} + (i ∗ w_e + r.l) ∗ w_{e_i} AS l,
            x.r − i ∗ w_{e_i} + (i ∗ w_e + r.l) ∗ w_{e_i} AS r
  FROM   I, T_{e_i} x, ROOTS(T_e) r
  WHERE  i ∗ w_{e_i} ≤ r.l AND r.r < (i + 1)w_{e_i}
    AND  i ∗ w_{e_i} ≤ x.l AND x.r < (i + 1)w_{e_i}
```

On this representation we define $T'_{e'}$ inductively. The result of the iterator is then

```
CREATE VIEW T_{for x∈e do e'}  AS  SELECT * FROM T'_{e'}
```

where $w_{\mathsf{for}\, x \in e \,\mathsf{do}\, e'} = w_e w_{e'}$. Note that $T_{\mathsf{for}\, x \in e \,\mathsf{do}\, e'}$ differs from $T'_{e'}$ only in its *width* (which is adjusted as a result of the concatenation of the results of the individual iterations of the $\mathsf{for}\, x \in e \,\mathsf{do}\, e'$ iterator).

Figure 6 illustrates the four steps involved in the translation of a $\mathsf{for}\, x \in e \,\mathsf{do}\, e'$ expression: (1) the translation of the subexpression e, (2) the extension of the original environment, (3) the translation of e' with respect to this new environment, and (4) the return to the original environment.

Example 4.3 Continuing with Example 3.4, consider the first iteration construct in query Q8:

```
for $p in document("auction.xml")/site/people/person
```

Below we show the SQL needed to generate I' and T'_p, which encode the variable `$p` *inside* the `for` loop, from the relations in Figure 5.

```
CREATE VIEW ROOTS_person AS
  SELECT u.s AS s, u.l AS l, u.r AS r
  FROM T_person u
  WHERE NOT EXISTS (
     SELECT *
     FROM T_person v
     WHERE v.l < u.l AND u.r < v.r)

CREATE VIEW I' AS
  SELECT i*86 + r.l AS i
  FROM I, ROOTS_person r
  WHERE i*86 <= r.l AND r.r < (i+1)*86

CREATE VIEW T_p' AS
  SELECT s,
     l - i*86 + (i*86 + r.l)*86 AS l,
     r - i*86 + (i*86 + r.l)*86 AS r
  FROM T_person x, ROOTS_person r, I
  WHERE i*86 <= r.l AND r.r < (i+1)*86
    AND r.l <= x.l AND x.r <= r.r
```

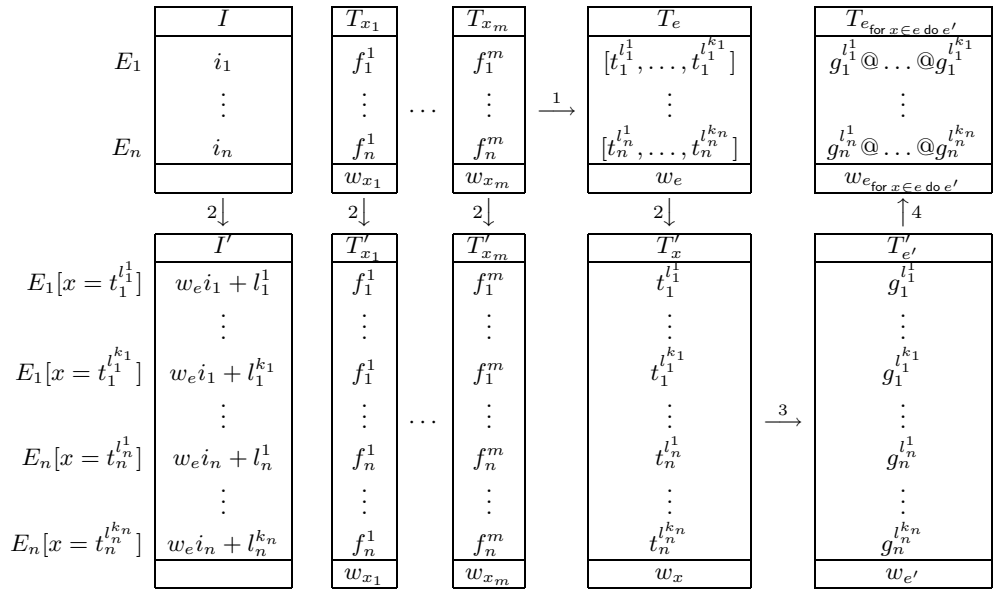

Figure 6: Translation of "$\mathsf{for}\, x \in e\, \mathsf{do}\, e'$" in the Environment for $x_1, \ldots, x_m$.

I'	T'_{p}		
i	s	l	r
2	<person>	174	195
	@id	175	178
	person0	176	177
	<name>	179	182
	Jaak Tempesti	180	181
	<emailaddress>	183	186
	mailto:Tempesti@labs.com	184	185
	<phone>	187	190
	+0 (873) 14873867	188	189
	<homepage>	191	194
	http://www.labs.com/~Tempesti	192	193
24	<person>	2088	2109
	@id	2089	2092
	person1	2090	2091
	<name>	2093	2096
	Cong Rosca	2094	2095
	<emailaddress>	2097	2100
	mailto:Rosca@washington.edu	2098	2099
	<phone>	2101	2104
	+0 (64) 27711230	2102	2103
	<homepage>	2105	2108
	http://www.washington.edu/~Rosca	2106	2107
	$w_p = 86$		

Figure 7: Persons in Separate Environments

The relations I' and T'_{p} generated by this SQL are shown in Figure 7. The two entries in the new index table I' correspond to the two roots of the XML forest encoded in $T_{\texttt{person}}$. Each new value of i, representing a separate environment inside the loop, indexes into a non-overlapping interval within T'_{p}: $i = 2$ corresponds to the range $[2 * w_{\mathrm{p}}, (2+1) * w_{\mathrm{p}} - 1] = [2 * 86, 3 * 86 - 1] = [172, 257]$, while $i = 24$ corresponds to [2064, 2149]; these are guaranteed to bracket the intervals of the corresponding persons in Figure 7.$\diamond$

4.3 Properties of the Translation

Using induction on structure of FLWR expressions, the above cases yield the correctness proof for the translation.

Proposition 4.4 *Let $I, T_{x_1}, \ldots, T_{x_m}$ be a representation of the initial environment E, and let e be a FLWR expression over $x_1, \ldots, x_m$. Then I, T_e defined by the translation given in Sections 4.1, 4.2.1, 4.2.2, 4.2.3, and 4.2.4 represents $[\![e]\!] E$.*

Results relating to the expressive power characterizing the FLWR expressions as well as the translation described above are presented in [34].

An additional benefit of being able to translate XQuery (as defined in Definition 2.2) to SQL (first-order logic with ordering, range-restricted arithmetic, and counting) is an automatic guarantee of polynomial time query evaluation algorithms. In addition, existing relational systems provide solid foundations for implementation of an XQuery processor based on this translation.

From an implementation point of view, an essential property of the translation is that the values of the interval encoding produced during the execution of the I and T_e queries are bounded by a polynomial in the size of the values in the encoding of the inputs. The degree of the polynomial depends only on the size (depth of nesting) of the expression e. While the bound on the width of the intervals is a theoretically satisfactory result, a practical implementation will map the interval endpoints to a (usually fixed) machine representation of integers. Using the above property, the translation can always allocate sufficient number of integer-valued attributes *at query compilation time* to hold the values of the left and right endpoints of the intervals at every stage of evaluation of the given expression.

5. IMPLEMENTATION STRATEGY

To make the XQuery translation practical, we need to examine if the queries produced by the translation are amenable to efficient relational execution strategies. Indeed, we find several patterns related to the essential use of *order predicates* in the translation that negatively affect perfor-

mance of standard relational systems. Thus, care is needed in implementing techniques that involve relational execution of XQuery expressions.

There are two important issues that need to be addressed in an efficient implementation. The first is the mapping of XQuery expressions to SQL such that ordering can be specified in the SQL queries. The second is, of course, the development of efficient execution strategies of the resulting queries. We discuss both of these issues in this section.

The maintenance of order in translating XQuery expressions to SQL queries is accomplished, in our approach, by the introduction of predicates in the SQL queries involving intervals. These queries can be executed by standard relational query processors, but performance is an issue. In particular, processing joins that relate tuples via possibly several *integer order conditions*, which are frequent in the translation of basic XML operators, commonly lead to unacceptable performance penalty when executed using a relational system. This fact has been recognized in the literature, and special-purpose operators have been developed to address these shortcomings [8, 36].

The prototype implementation of our approach follows a similar path. However, instead of focusing on special-purpose algorithms for the individual operations, we consider the *interaction* of such operators with standard relational plan operators. Indeed, many of the basic XQuery functions can be translated to *efficient* relational plans without special-purpose operators[3].

A distinguishing characteristic of the special-purpose operators introduced below is that they expect their inputs *sorted* by the *left endpoint* (or, alternatively, the right endpoint) of the interval values in the encoding. The operators also maintain the same ordering in the outputs, usually running in linear time. This property allows for the creation of query execution plan fragments that involve a sequence of linear time operations. While this is possible for most of the queries that implement the *basic* operations on XML forests (cf. Figure 2), when the situation calls for a differently ordered representation the cost of reorderings is still $O(n \log n)$ time (versus falling into quadratic time operations, such as a nested loop joins).

With respect to efficient implementation of the resulting SQL queries, we develop new physical plan operators (e.g., tree merge-join) and we ensure that the generated SQL queries are amenable to optimization using these new operators. The main thrust of our approach is to be able to create plans that employ linear (merge) operations that take advantage of the order of the relational representation (which often coincides with the document order) while reusing all the remaining physical operators already developed for relational engines.

Example 5.1 Consider the query that defines the `ROOTS` operation, given in Section 4.1. A generic relational system takes $O(n^2)$ time in worst case as the `NOT EXISTS` clause cannot be resolved due to the presence of *two* order selection conditions.$\diamond$

This query can be implemented by an efficient plan by adding the operation described below to the repertoire of physical operators of the relational system.

[3]This is why we do not have to discuss the translations for all the primitive operations in Figure 2.

Algorithm 5.2 (Roots Extraction) Given the precondition that the tuples in the input `Iterator T` are ordered by the `l` value.

```
Iterator Roots(Iterator T) {
  int currentRootRight=0; // distance covered by current root
  Tuple TT;  // tuples for environment and tuples

  Tuple fetch() {
    while (true) {
      TT = T.fetch();
      // no more tuples
      if (TT==null)  return END-OF-INPUT;
      // next root?
      if (TT.l>currentRootRight) {
        currentRootRight = TT.r;
        return TT;
      } // otherwise it's a child; loop
    }
  }
}
```

The algorithm runs in time $O(n)$ and space $O(1)$ in the number of tuples on the input iterator.

Another important primitive operation for which a special-purpose operator is needed is the *structural comparison* that compares sets of tuples that encode two forests to decide their relative (structural) ordering. Although deep comparison can be expressed in SQL with counting, the resulting expression is impractical enough to justify a primitive physical operator.

Algorithm 5.3 (Deep Comparison) Given two iterators that produce tuples sorted by their `l` values, the following procedure decides which of the iterators contains the smaller tree (in structural order).

```
Comparison DeepCompare(Iterator L,R) {
  // returns LESS, EQUAL, or GREATER
  // depending on how forests L and R compare

  Stack ST;    // Stack of pairs (lr,rr) of integers
  Tuple TL,TR; // Holders of (s,l,r) triples

  while (true) {
    // fetch next pair of tuples
    TL = L.fetch();   TR = R.fetch();
    // no more tuples
    if (TR==null)&&(TR==NULL) return EQUAL;
    // ''smaller'' forest done
    if (TL==null)             return LESS;
    if (TR==null)             return GREATER;
    // check if the nodes represented by these
    // two tuples fall in the same place in the
    // two forests
    while (ST.nonEmpty())&&
          ((TL.r<ST.top().lr)||(TR.r<ST.top().rr)) {
      // check for a ''missing sibling''
      if (TL.r>ST.top().lr)   return LESS;
      if (TR.r>ST.top().rr)   return GREATER;
      // otherwise the nodes match so far
      s.pop();
    }  // positions match--order by label:
    if (TL.s<TR.s)            return LESS;
    if (TL.s>TR.s)            return GREATER;
    // everything matches so far, so repeat
    ST.push(TL.r,TR.r)
  }
}
```

The algorithm runs in time linear in the size of the XML forests and the required stack size is bounded by the *depth* of the encoded forests.

Additional physical operators added to the relational system to support evaluation of path navigation and structural equality are, however, not sufficient for efficient execution of XQuery expressions. To achieve performance comparable to that of relational systems, the *nested-loop* evaluation of FLWR "$\mathsf{for}\, x \in e\, \mathsf{do}\, e'$" expressions must be avoided (as demonstrated by experiments in Section 6).

The simplest example of this situation may look as follows:

$$\mathsf{for}\, x \in e_1(z)\, \mathsf{do}\, \mathsf{for}\, y \in e_2(z)\, \mathsf{do}\, \mathsf{where}\, x = y\, \mathsf{return}\, e,$$

where the expression e_2 does not depend on the variable x. In this situation our translation recognizes that the two loops can be executed *independently* using only data supplied by the (encoding of the) enclosing environment to supply the value for the variable(s) z. Applying the translation for the FLWR iterator on the encoding of an input environment, I, T_z, independently yields two environments: I^x, T_z^x, T_x^x and I^y, T_z^y, T_y^y (superscripted by the respective variable name introduced by the iterator). It is easy to see that the I^x, T_x^x and I^y, T_y^y components of the environments contain sufficient information to resolve the conditional in the expression. While this again can be done by a SQL query, an efficient approach requires sorting the environment indices I^x and I^y with respect to the structural ordering of the forests represented in the T_x^x and T_y^y tables, respectively. We utilize the efficient algorithm for determining structural order of XML forests here. The results can then be merged using the structural comparison in a single pass (similar to a relational merge-join algorithm). The result of this operation is a set of pairs of matching environment indices. These are then used to construct the environment $I^{xy}, T_z^{xy}, T_x^{xy}, T_y^{xy}$ needed as an input to the translation of the expression e. Note that this environment is identical to the environment we would have obtained using the nested-loop strategy; thus all the properties of the translation, in particular the document ordering, are preserved as required by the semantics of XQuery.

While the above example exposes the main idea behind our merge-join approach to executing FLWR expressions, the actual transformation is more involved. Consider again (a fragment of) our running example:

```
for $p in document("auction.xml")/site/people/person
let $a := (for $t in document("auction.xml")/site/
               closed_auctions/closed_auction
           where $t/buyer/@person = $p/@id
           return $t)
 ...
```

While the two `for` loops in this example do not follow the above pattern exactly, our translation still recognizes that the *inner* loop can be executed independently of the outer loop and produces a relational execution plan that follows the above approach. Moreover, the handling of environments in the translation of the inner loop and the assignment of its result to the variable `$a` automatically yields the *outer join*-like behavior of the outer loop (i.e., `person`s appear in the output even if they haven't participated as a `buyer` in any `closed_auction`s)[4].

[4]Full details of the transformations are omitted here due to space restrictions.

Note that while we mainly focus on operators to support relational queries obtained by translating XQueries, the special-purpose operators developed here are *relational operators* that correspond to patterns in SQL queries (mainly relating to order of values present in the answers to the queries). Such patterns are present in many other situations where ordering of values is essential, e.g., in *temporal queries* [30, 33].

6. EXPERIMENTAL RESULTS

In this section, we present the results of experiments that demonstrate the value of the dynamic interval approach described in this paper. Although we would ideally use an existing relational database system for this purpose, as described in Section 5 and in other literature, existing commercial relational engines are not tuned to efficiently handle interval processing. Instead, we demonstrate our approach using a prototype relational engine, written in Java, which was extended with the specialized relational operators described in Section 5. We claim that the positive performance results obtained using on this prototype are easily transferable to a similarly enhanced commercial relational system.

We will refer to our prototype as DI (for *Dynamic Intervals*) for the remainder of this section. XQueries were translated to a relational algebra-style query plan which implements the SQL query generated using the translation rules detailed in Section 4. This plan was then executed by the DI prototype. We must stress that these translations were generic; although our query plans utilize our proposed specialized operators where appropriate, no schema-based rewriting or cost-based optimization was performed.

We present here a comparison of the performance of the DI prototype with several XQuery processors and native XML database systems, including: *Galax* v0.2.0 from Lucent Technologies Inc. [1]; *IPSI-XQ* v1.1.1b from Fraunhofer IPSI [2]; *Kweelt* from Sourceforge [3]; *QuiP* from Software AG [4][5]; and *X-Hive/DB* v4.0 from X-Hive Corporation [5].

We have used XML documents generated by the XMark benchmark [6] with five scale factors, 0.001 (113kB), 0.01 (1.11MB), 0.1 (11.1MB), 1 (111MB), and 10 (1.09GB). Experiments were run on a dual-1.0GHz Pentium III system with 1GB RAM running RedHat Linux 7.1. All numbers reported are the average of the combined user and system CPU times over five executions. Any query evaluation that did not finish within two hours of CPU time (denoted "DNF" for "Did Not Finish" in charts) or whose memory demands exceeded the capacity of the experimental sysem (denoted "IM" for "Insufficient Memory" in charts) was terminated. Our query timings do not include any initial document load times for those systems that required it.

Note that indexing of XML documents is orthogonal to the techniques proposed in this paper. Indices commonly only apply to leaf query operators, while our approach is applicable throughout the execution of a query, including constructed intermediate results. For simplicity, we do not define any indices in our experiments for any of the systems.

6.1 Result Construction

Our first experiment tests the ability of a system to reconstruct large portions of the original document, for which

[5]Note that in our experiments QuiP read documents from the file system, not from a Tamino database.

we used XMark query Q13:

```
for $i in document("auction.xml")/site
                      /regions/australia/item
return
  <item name="{$i/name/text()}"> $i/description </item>
```

Timing results for Q13 are shown in Figure 8. DI and X-Hive

System	XMark scale factor				
	0.001	0.01	0.1	1	10
Kweelt	1.3	2.4	9.0	94.6	IM
IPSI-XQ	2.4	3.3	6.8	45.7	IM
Galax	0.1	0.6	6.4	106	IM
QuiP	0.6	5.8	62.7	IM	IM
X-Hive	4.3	4.4	5.0	9.4	51.6
DI	0.4	0.7	2.4	18.2	276

Figure 8: Q13 Timings (CPU sec)

are the only systems that scale up to large database sizes (almost linearly). Although our system is based on a generic shredding of the document into relations, it is competitive with a commercial system optimized for processing XML documents. The other systems were not able to handle large documents.

6.2 A Single Join Query

Our main experiment used query Q8 from the XMark benchmark (with a minor modification):

```
for $p in document("auction.xml")/site/people/person
let $a := for $t in document("auction.xml")/site
                      /closed_auctions/closed_auction
          where $t/buyer/@person = $p/@id
          return $t
where exists($a)
return <item person="{$p/name/text()}">count($a)</item>
```

The query is analogous to a relational join between two tables, followed by a `"group by"` operation on the columns of one of the tables. Our modification essentially converts an outer- to an inner-join, which minimizes the size of the results and better isolates the time spent evaluating the join. We created two different query plans for evaluating Q8 with the DI system; the only difference between them was that where one plan used a nested-loop join operator (DI-NLJ), the other used a merge-sort join and then sorted by the dynamic intervals to re-establish document order (DI-MSJ).

System	XMark scale factor				
	0.001	0.01	0.1	1	10
Kweelt	2.6	102	DNF	DNF	DNF
IPSI-XQ	2.8	12.2	784	DNF	DNF
Galax	0.1	3.3	285	DNF	DNF
QuiP	13.6	DNF	DNF	DNF	DNF
X-Hive	4.4	5.3	80	DNF	DNF
DI-NLJ	2.5	4.7	263	DNF	DNF
DI-MSJ	2.3	2.9	6.7	47.3	866

Figure 9: Q8 Timings (CPU sec)

Figure 9 shows timings for Q8 from the five other XQuery systems as well as from our prototype using the two different join plans.

All of the other systems scaled similar to DI-NLJ, which is a quadratic query plan that fails to evaluate the 111MB document within the two hour time limit. In contrast, DI-MSJ easily evaluates Q8 on the 1GB file. Figure 10 shows a breakdown of the CPU time for DI-NLJ and DI-MSJ among the plan components (path extraction, evaluation of the join, and construction of results). While the join operator accounts for almost all of the cost of the nested-loop evaluation of the 11MB document, even for the largest document the MSJ plan incurred most of its cost not on the join, but on evaluating path expressions. Because paths were evaluated using sequential scans of the relation, this cost could be significantly reduced in most cases by the use of indices. These results illustrate the importance of providing an alternative to nested-loop evaluation for nested iteration constructs.

System	Component	XMark scale factor				
		0.001	0.01	0.1	1	10
DI-NLJ	Paths	32%	25%	1%	<1%	
	Join	23%	49%	98%	>99%	
	Construction	45%	26%	1%	<1%	
DI-MSJ	Paths	35%	41%	61%	71%	68%
	Join	16%	16%	12%	13%	27%
	Construction	49%	43%	27%	15%	5%

Figure 10: Q8 Timing Breakdown

Plans DI-NLJ and DI-MSJ were written using join operators that perform structural equality comparisons described in Section 5. However, in this query the join keys are simple attribute values, so our specialized join operators function equivalent to traditional relational NLJ and MSJ operators. In fact, in any case where either the XQuery itself or schematic information allows us to deduce a fixed representation for the join keys, we can use traditional relational join operators to evaluate the join. In a separate experiment (not shown for space reasons), we replaced the attribute join keys with elements containing trees of varying depth and fanout and verified that the costs of structural-equality join operators grow linearly with the number of nodes in the join key. Several of the other systems we tested failed in this experiment because they were not able to correctly test for structural equality of XML documents.

6.3 A Multiple Join Query

Our final experiment uses (a similarly modified) Q9 from the XMark benchmark:

```
for $p in document("auction.xml")/site/people/person
let $a := for $t in document("auction.xml")/site
                      /closed_auctions/closed_auction
          let $n := for $t2 in document("auction.xml")
                              /site/regions/europe/item
                    where $t/itemref/@item = $t2/@id
                    return $t2
          where $t/buyer/@person = $p/@id
          and exists($n)
          return <item> $n/name/text() </item>
where exists($a)
return <person name="{$p/name/text()}"> $a </person>
```

Unlike Q8, whose aggregate function actually permits violation of document order for the inner FLWR expression, Q9 places document order constraints on all three levels of iteration.

System	XMark scale factor				
	0.001	0.01	0.1	1	10
Kweelt	11.9	DNF	DNF	DNF	DNF
IPSI-XQ	3.3	14.9	1033	DNF	DNF
Galax	0.4	170	DNF	DNF	DNF
QuiP	141	DNF	DNF	DNF	DNF
X-Hive	4.4	4.8	94.5	DNF	DNF
DI-NLJ	3.5	5.5	123	DNF	DNF
DI-MSJ	3.4	4.2	9.7	69.6	1178

Figure 11: Q9 TIMINGS (CPU SEC)

An interesting feature of this query is that although it has one more join than Q8, the join between `auctions` and `items` is a highly selective foreign-key-like join. Figure 11 shows timing results for Q9. As with Q8, we tested our own system using both a NLJ-based plan and a MSJ-based plan. Kweelt, Galax, and QuiP exhibit timings for Q9 much worse than for Q8. In contrast, IPSI-XQ, X-Hive, DI-NLJ, and DI-MSJ all scale similar to their Q8 performance. This experiment demonstrates that the performance advantage of our approach carries over to arbitrary nesting of FLWR loops.

7. RELATED WORK

The use of an interval encoding for dealing with hierarchical queries in the relational world is well known [15]. The representation has been used, in the area of document management, for specifying containment in text databases [20, 21] and for manipulating SGML data in a relational system [12]. More recently, the performance behavior of interval processing within relational systems has been studied in the context of XML containment queries [8, 36]. The approaches, however, do not address *dynamic* aspects of XQuery evaluation that are linked to nested iteration (for-loops) and to creation of new documents.

Other relational encodings of XML documents to relations have been studied [25]. Recently, encodings that preserve document order have been investigated [31, 11] in the context of the XPath fragment of XQuery. However, none of the approaches addresses mapping of FLWR expressions to relational queries.

The problem of mapping XPath queries to SQL queries over the interval encoding has been studied in [24, 35]. The approaches deal with *data retrieval* only and do not address evaluation of more complex XQuery expressions.

The only translation that considers FLWR expressions over arbitrary XML documents was proposed in [26]. However, it uses an edge-based encoding of XML documents and relies on a transitive closure relation to encode document hierarchy with a potential $O(n^2 \log n)$ bits storage cost (compared to the $O(n \log n)$ bits storage cost needed by our approach).

The edge-based encoding forces the translation to use multiple SQL expressions connected via auxiliary code in a general-purpose programming language (or non first-order features present in the upcoming SQL standard, since transitive closure cannot be maintained using first-order queries). This potentially limits the opportunities for query optimization. In contrast, our approach produces a *single SQL statement* for arbitrarily-nested FLWR expressions, including the construction of *new nested documents*, possibly in the scope of other FLWR expressions, while seamlessly preserving the document order required by XQuery semantics.

Also, since the edge-based encoding does not explicitly capture information about document order, the results of the translation are limited to *nested-loop* based operations in an attempt to preserve document order [26]. Since our approach captures and manipulates information about document order explicitly, the generated relational queries are not limited to such inferior evaluation strategies. Indeed, experimental results presented in Section 6 corroborate that the preferable algorithms enabled by our translation yield performance advantage of several orders of magnitude.

The techniques proposed in this paper (and in [26]) target arbitrary XML documents. Thus, our techniques do not rely upon the availability of fixed (essentially *relational*) schematic information that is essential to approaches for processing XQueries over XML data "*published*" from relational databases [14, 24, 28, 23].

Finally, recent work on physical operators provides efficient support of various fragments of XPath [8, 13, 36]. Our approach can utilize these operators in the query plans. We also extend the repertoire of such operators by introducing an efficient implementation for structural comparisons of XML forests encoded using the interval representation. Such an operation is essential to support merge-join evaluation strategies in XQuery.

8. CONCLUSION

We have introduced *dynamic intervals*, an encoding based on an interval representation of XML documents that supports predictably good query plans for executing arbitrarily nested XQuery FLWR expressions.

We have presented a translation of XQuery expressions drawn from a *comprehensive* subset of XQuery to *single* SQL statements. This translation fully supports arbitrary combinations and nesting of basic functions, XPath expressions and axes, element constructors, structural equality, and nested FLWR expressions, sorting, etc., without the need for an escape to a general-purpose programming language. The plans produced by the translation faithfully capture the semantics of XQuery and, in particular, maintain the required document order.

In addition, our analysis and experimental results show that the dynamic interval-based plans scale (almost) linearly, enabled by merge-join evaluation strategies. Experimental results presented in the paper show the performance advantage gained by our approach over quadratic behavior exhibited by a number of other XQuery systems.

While the XQuery translation described in this paper targets a relational implementation (in particular based on extending existing relational engines), many of the proposed techniques and strategies can also be used within the context of native XML systems.

Acknowledgment

We would like to thank H. V. Jagadish and Laurent Mignet for their comments on earlier versions of this document as well as X-Hive Corporation for their cooperation and comments. The authors gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada for support of this research.

9. REFERENCES

[1] Galax. Available from http://db.bell-labs.com/galax/.

[2] IPSI-XQ - the XQuery demonstrator. Available from http://ipsi.fhg.de/oasys/projects/ipsi-xq/.

[3] Kweelt. Available from http://kweelt.sourceforge.net.

[4] QuiP. Available from http://developer.softwareag.com/tamino/quip/.

[5] X-Hive/DB. Available from http://www.x-hive.com.

[6] XMark – an XML benchmark project. Available from http://www.xml-benchmark.org.

[7] V. Aguilera, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying XML documents in Xyleme. In *Proc. ACM SIGIR Workshop on XML and Information Retrieval*, 2000.

[8] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. 18th Intl. Conf. on Data Engineering*, 2002.

[9] D. Barbosa, A. Barta, A. O. Mendelzon, G. A. Mihaila, F. Rizzolo, and P. Rodriguez-Guianolli. ToX - The Toronto XML Engine. In *Proc. Workshop on Information Integration on the Web*, pages 66–73, 2001.

[10] S. Boag, D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML Query Language. Technical report, W3C, 2001.

[11] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From xml schema to relations: A cost-based approach to xml storage. In *Proc. 18th Intl. Conf. on Data Eng.*, pages 64–75, 2002.

[12] L. J. Brown, M. P. Consens, I. J. Davis, C. R. Palmer, and F. W. Tompa. Structured Text ADT for object-relational databases. *Theory and Practice of Object Systems*, 4(4):227–244, 1998.

[13] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proc. SIGMOD Conference*, pages 310–321, 2002.

[14] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *Proc. 26th Intl. Conf. on Very Large Data Bases*, pages 646–648, 2000.

[15] J. Celko. Trees, Databases and SQL. *DBMS*, 7(10):48–57, 1994.

[16] D. Chamberlin. XQuery: An XML query language. *IBM Systems Journal*, 41(4):597–615, 2002.

[17] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. XML Query Use Cases. Technical report, W3C, 2001.

[18] Y. Chen, G. Mihaila, S. Padmanabhan, and R. Bordawekar. Labeling Your XML. (preliminary version presented at CASCON'02 Conf.), October 2002.

[19] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *Proc. 21st PODS Symposium*, pages 271–281, 2002.

[20] M. P. Consens and T. Milo. Optimizing queries on files. In *Proc. SIGMOD Conference*, pages 301–312, 1994.

[21] M. P. Consens and T. Milo. Algebras for querying text regions. In *Proc. 14th PODS Symposium*, pages 11–22, 1995.

[22] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 Formal Semantics. Technical report, W3C, 2001.

[23] M. Fernndez, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. Silkroute: A framework for publishing relational data in xml. *ACM Trans. Database Syst.*, 27(4):438–493, 2002.

[24] T. Fiebig and G. Moerkotte. Evaluating Queries on Structure with eXtended Access Support Relations. In *Proc. WebDB Workshop*, volume 1997 of *Lecture Notes in Computer Science*, pages 125–136, 2000.

[25] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[26] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries over Heterogenous Data Sources. In *Proc. 27th Intl. Conf. on Very Large Data Bases*, pages 241–250, 2001.

[27] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: a database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[28] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proc. 26th Intl. Conf. on Very Large Data Bases*, pages 65–76, 2000.

[29] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. 25th Intl. Conf. on Very Large Data Bases*, pages 302–314, 1999.

[30] R. T. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Kafer, N. Kline, K. Kulkarni, T. Y. C. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. A. Sripada. TSQL2 language specification. *SIGMOD Record*, 23(1):65–86, Mar. 1994.

[31] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. SIGMOD Conference*, pages 204–215, 2002.

[32] F. Tian, D. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative XML storage strategies. Technical report, University of Wisconsin, 2002.

[33] D. Toman. Point-based temporal extensions of SQL. In *Proc. Intl. Conf. on Deductive and Object-Oriented Databases*, pages 103–121, 1997.

[34] D. Toman and G. E. Weddell. Querying XML: On the Utility of Interval Encoding. Technical report, University of Waterloo, 2002.

[35] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transactions on Internet Technology*, 1:110–141, 2001.

[36] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. SIGMOD Conference*, pages 425–436, 2001.